Time Series Forecasting of Carbon Dioxide Emissions by Electricity Generation in USA.

```
!pip install statsmodels==0.12.2
from math import sqrt
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import pyplot
import seaborn as sns
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
from statsmodels.tsa.stattools import kpss
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.seasonal import STL
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.api import SimpleExpSmoothing
from sklearn.metrics import mean_squared_error
import warnings
warnings.filterwarnings("ignore")
```

```
Requirement already satisfied: statsmodels==0.12.2 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: pandas>=0.21 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: patsy>=0.5 in /usr/local/lib/python3.7/dist-packages (fro
Requirement already satisfied: scipy>=1.1 in /usr/local/lib/python3.7/dist-packages (fro
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.7/dist-packages (f
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from patsy
```

```
#Converting the provided into timeseries data by parsing the indexes to date time
dateparse = lambda x: pd.to_datetime(x, format='%Y%m', errors = 'coerce')
df = pd.read_csv("/content/MER_T11_01-1985.csv", parse_dates=['YYYYMM'], index_col='YYYYMM',
```

```
#shape of dataset
print(df.shape)
df.head()
```

```
(465, 4)
```

YYYYMM	MSN	Value	Description	Unit
1985-01-01	NNTCEUS	113.099	Natural Gas, Excluding Supplemental Gaseous Fu...	Million Metric Tons of Carbon Dioxide
1985-02-01	NNTCEUS	115.891	Natural Gas, Excluding Supplemental Gaseous Fu...	Million Metric Tons of Carbon Dioxide

```
emission_sources= df['Description'].unique()
```

03-01 Gaseous Fu... Carbon Dioxide

```
#High level statistics of numerical attributes
print(df.describe())
#Features in dataset
print(df.columns)
```

```
             Value
count   465.000000
mean    187.369497
std     301.947493
min      50.210000
25%      85.110000
50%     102.297000
75%     125.362000
max    1693.720000
Index(['MSN', 'Value', 'Description', 'Unit'], dtype='object')
```

```
#checking for null values
df['Value'].isna().sum()
```

```
0
```

```
#time series data after removing total energy emissions of each year
time_series= df[pd.Series(df.index).notnull().values]
print(time_series.shape)
```

```
(430, 4)
```

```
time_series.drop(['MSN','Description','Unit'],axis=1,inplace=True)
Energy_source_NaturalGas_New= time_series.copy()
time_series.head()
```

	Value
YYYYMM	
1985-01-01	**113.099**

```
#extracting month and year from data to create new features
Energy_source_NaturalGas_New['Year-month']= pd.to_datetime(time_series.index)
Energy_source_NaturalGas_New['Year']=Energy_source_NaturalGas_New['Year-month'].dt.year
Energy_source_NaturalGas_New['Month']=Energy_source_NaturalGas_New['Year-month'].dt.month
```

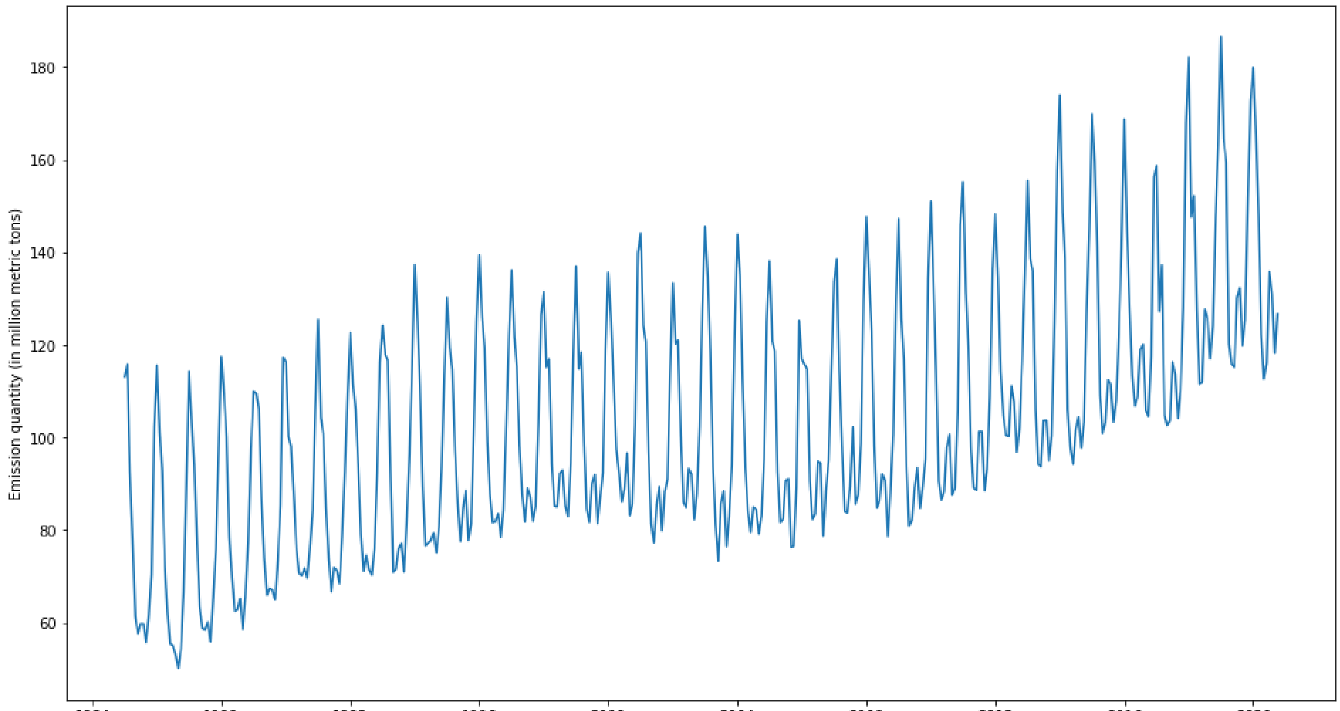
```
Energy_source_NaturalGas_New.head()
```

	Value	Year-month	Year	Month
YYYYMM				
1985-01-01	**113.099**	**1985-01-01**	**1985**	**1**
1985-02-01	**115.891**	**1985-02-01**	**1985**	**2**
1985-03-01	**92.664**	**1985-03-01**	**1985**	**3**
1985-04-01	**77.586**	**1985-04-01**	**1985**	**4**
1985-05-01	**61.334**	**1985-05-01**	**1985**	**5**

```
#Years of Co2 emission
print(Energy_source_NaturalGas_New['Year'].unique())
print(len(Energy_source_NaturalGas_New['Year'].unique()))
```

```
[1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998
 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012
 2013 2014 2015 2016 2017 2018 2019 2020]
36
```

```
fig = plt.figure(figsize = (16,9))
plt.plot(time_series.index,time_series['Value'])
plt.xlabel('Years')
plt.ylabel('Emission quantity (in million metric tons)')
plt.show()
```

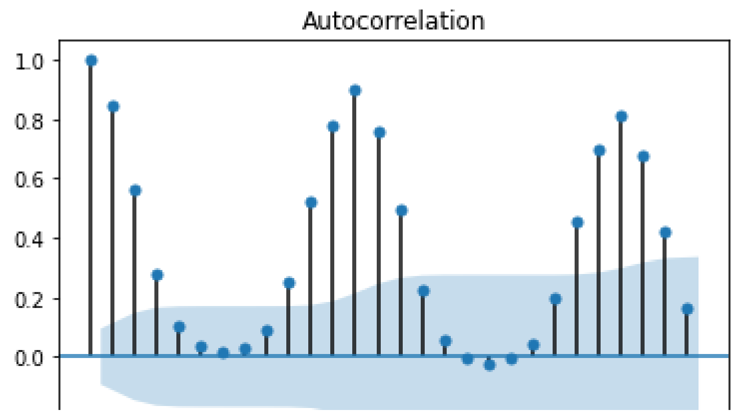


Conclusion:

- Increasing trend- Probably duet to increase in consumption of natural gas for electricity generation
- Seasonality with period(m=12)
- There are two seasonal peaks describing weather related fluctuations in energy demand.
- Seasonal fluctuations are constant.(Suggesting additive decomposition is favorable.)
- reason behind second small peak in a season?
- constant value of emissions between period 2000-2006, possibly due to energy/environment related global awareness and policies.

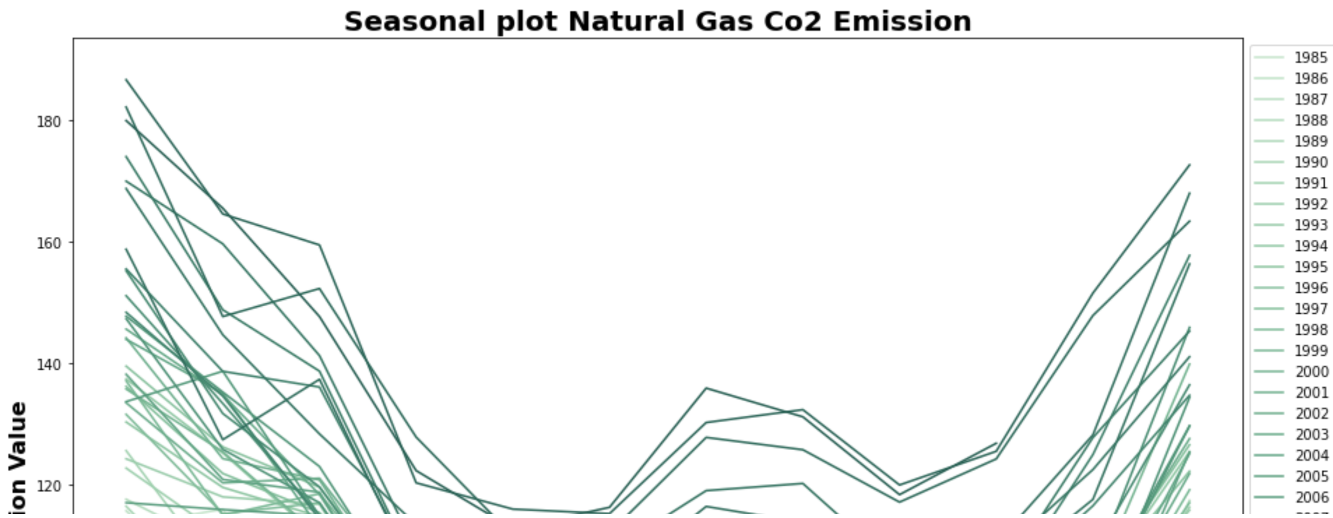
Correlogram - ACF Plot

```
def plotACF(ts):
  plot_acf(ts,alpha=0.05)
  pyplot.show()


def plotPACF(ts):
  plot_pacf(ts,alpha=0.05)
  pyplot.show()


plotACF(time_series['Value'])
```



- There is significant correlation between Yt and it lags.
- Seasonality of period(m=12) exists.
- Values of lag1, lag2, lag3... are also significant suggesting trend in time series.

```
fig, ax = plt.subplots(figsize=(15, 12))
palette = sns.color_palette("ch:2.5,-.2,dark=.3",36)
sns.lineplot(Energy_source_NaturalGas_New['Month'], Energy_source_NaturalGas_New['Value'], hu
ax.set_title('Seasonal plot Natural Gas Co2 Emission', fontsize = 20, loc='center', fontdict=
ax.set_xlabel('Month', fontsize = 16, fontdict=dict(weight='bold'))
ax.set_ylabel('Emission Value', fontsize = 16, fontdict=dict(weight='bold'))
plt.legend(bbox_to_anchor=(1, 1), loc=2)
plt.show()
```

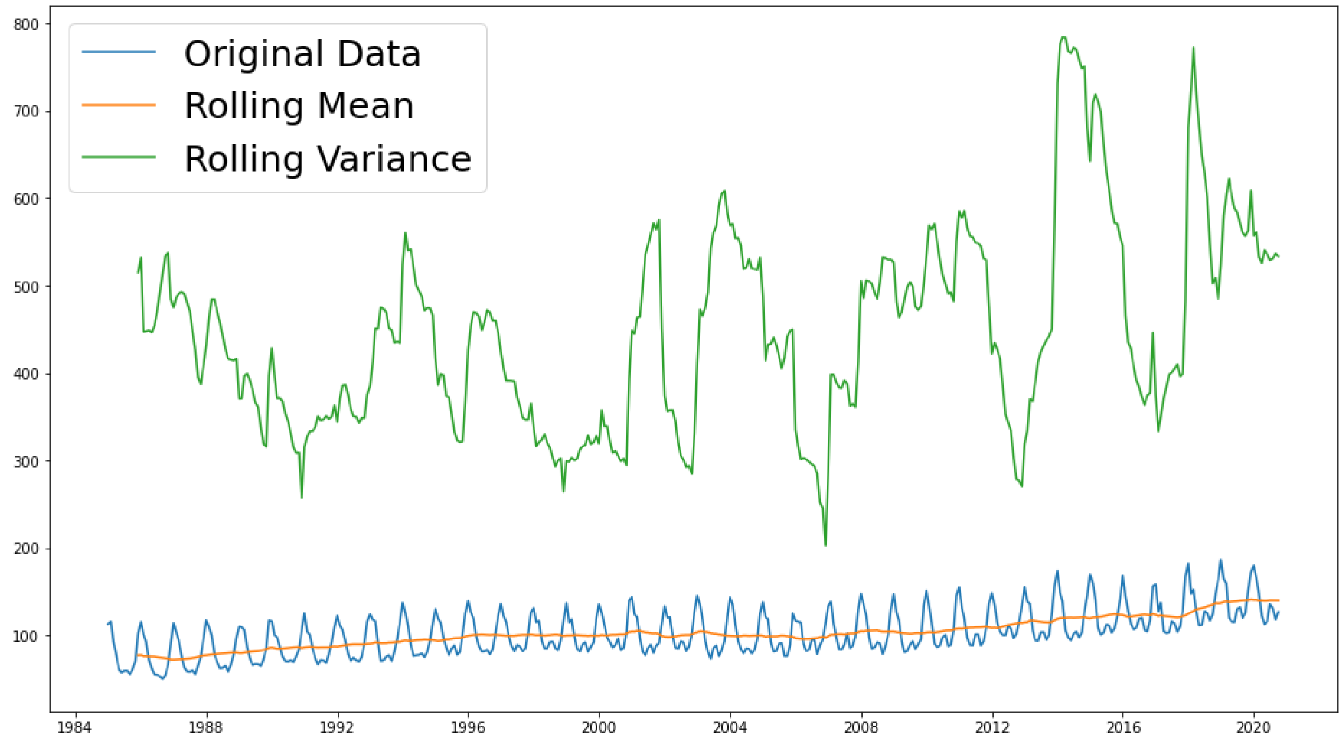


- This plot confirms our above finding of seasonality existence.
- The emission of Co2 is maximum in the year begining i.e in january possibly due to high electricity consumption for household and commercial purposes.
- The small peak arrives again between the month 6-8 i.e june- august, again due increase in electricity consumption during summer season.

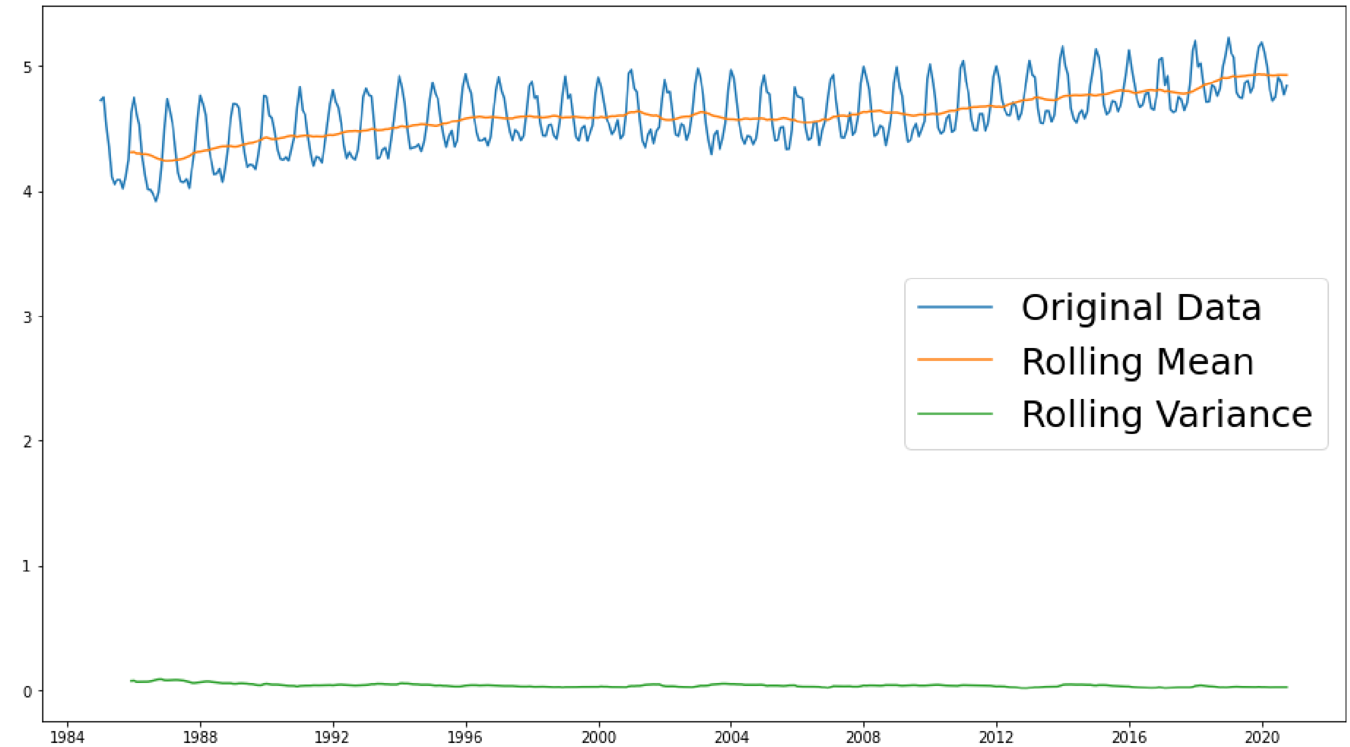
```
#Helper function to plot moving average
def check_stationarity_graphically(ts):
  t_s= pd.DataFrame()
  t_s['Rmean'] = ts.rolling(12).mean()
  t_s['Rstdv'] = ts.rolling(12).var()
  t_s['Rstdv'].replace(np.nan,0)
  t_s['Rmean'].replace(np.nan,0)
  fig = plt.figure(figsize = (16,9))
  plt.plot(time_series.index,ts,label='Original Data')
  plt.plot(time_series.index,t_s['Rmean'],label='Rolling Mean')
  plt.plot(time_series.index,t_s['Rstdv'],label='Rolling Variance')
  plt.legend(loc='best', fontsize = 25)
  plt.show()
```

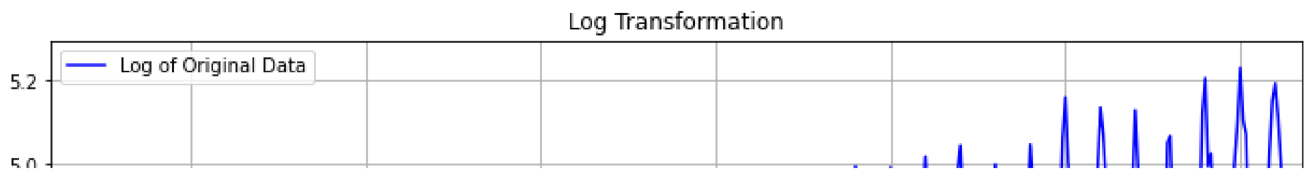
Transformation

```
#Plotting moving average and variance before transformation
check_stationarity_graphically(Energy_source_NaturalGas_New['Value'])
```



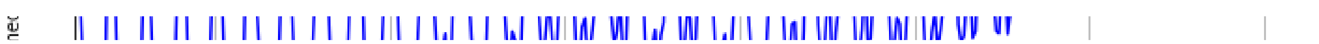

```
Energy_source_NaturalGas_New['Log'] = np.log(Energy_source_NaturalGas_New['Value'])
Energy_source_NaturalGas_New['Log'].replace(np.nan,0)
check_stationarity_graphically(Energy_source_NaturalGas_New['Log'])
Energy_source_NaturalGas_New['Log'].plot(figsize=(12,6), legend=True, label = 'Log of Origina
plt.show()
```



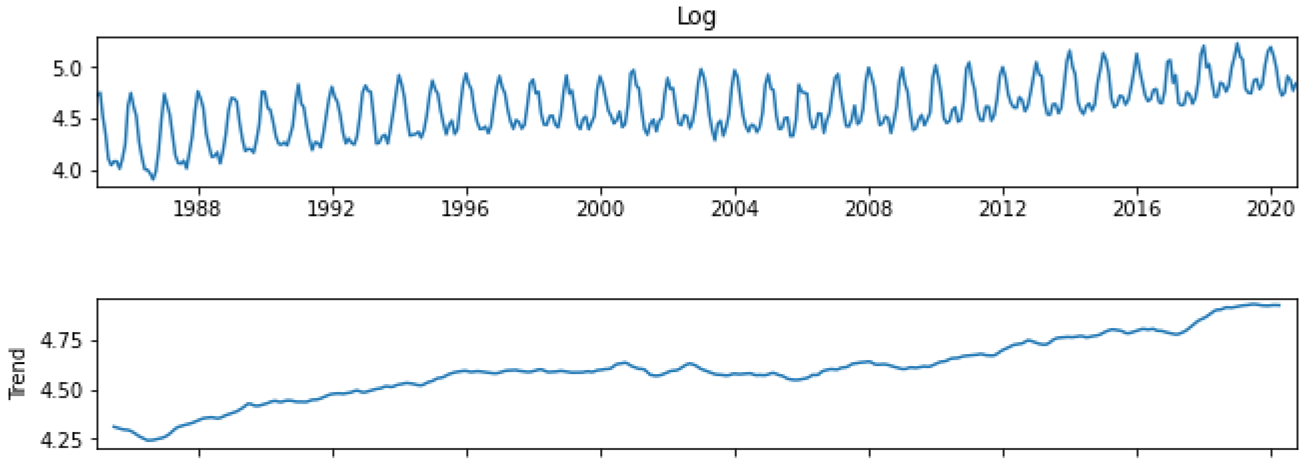


TimeSeries Decomposition

Classical Decomposition

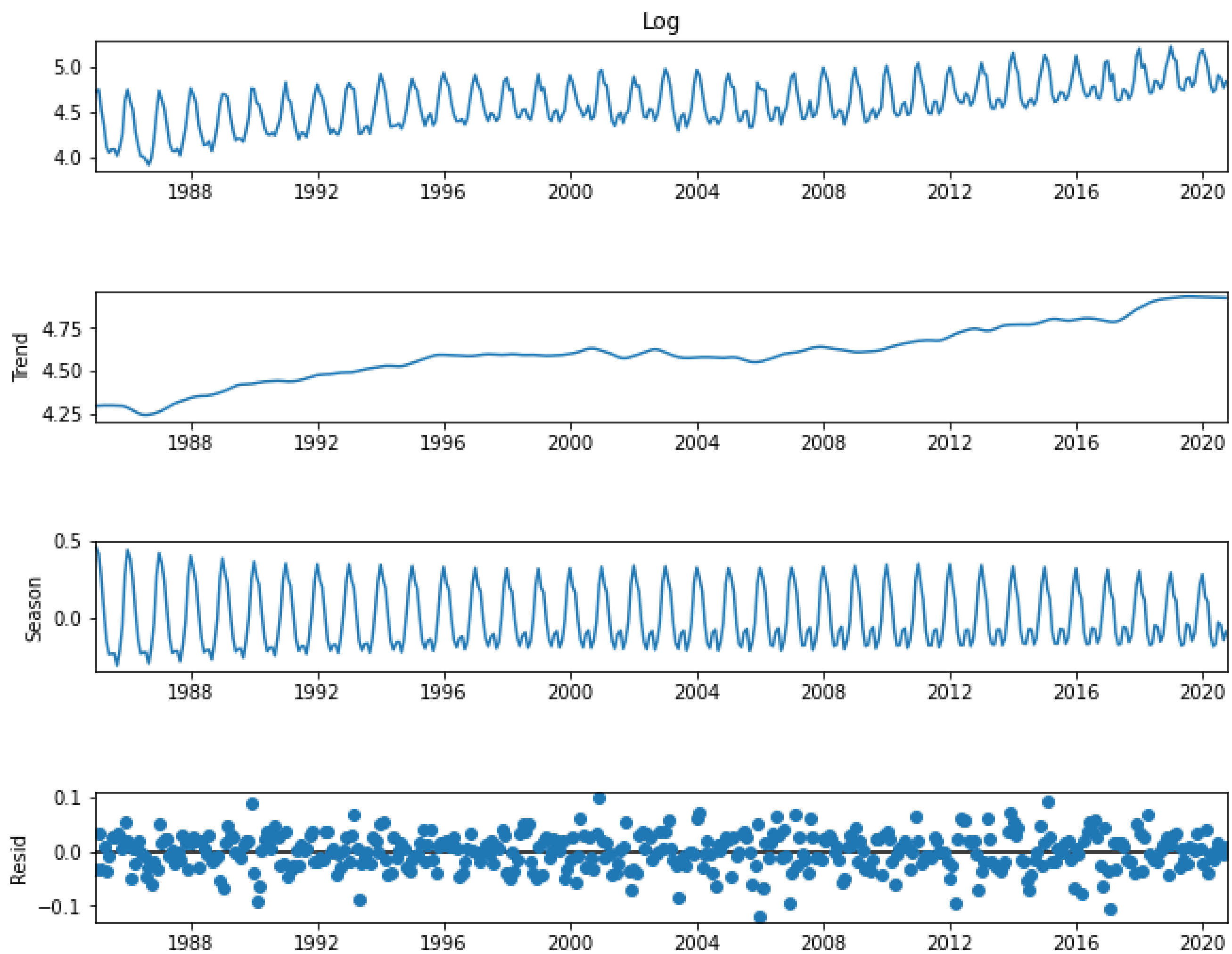


```
classical_decomposition = seasonal_decompose(Energy_source_NaturalGas_New['Log'], model='addi
fig = classical_decomposition.plot();
fig.set_size_inches(10,8)
plt.show()
```



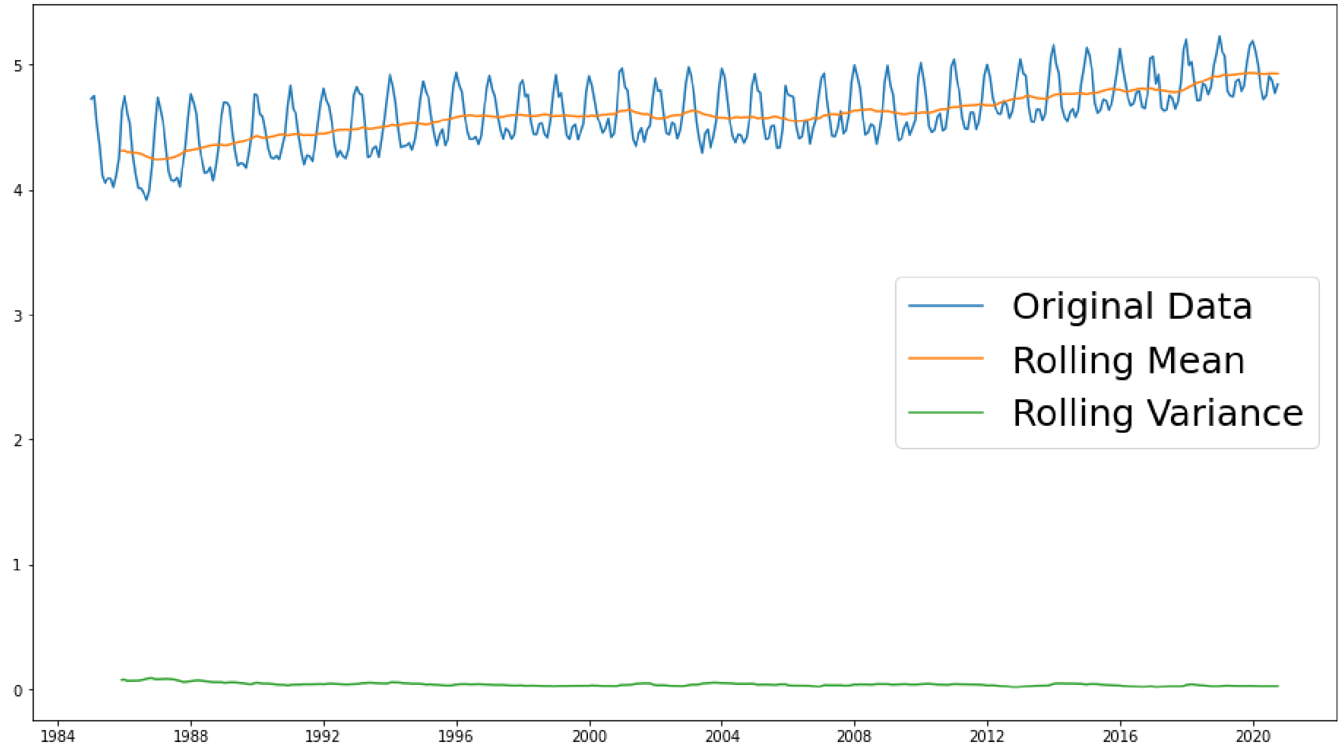
STL Decomposition

```
stl_decomposition = STL(Energy_source_NaturalGas_New['Log'], seasonal= 11)
res = stl_decomposition.fit()
fig = res.plot()
fig.set_size_inches(10,8)
plt.show()
```



Checking Stationary using Graph

```
check_stationarity_graphically(Energy_source_NaturalGas_New['Log'])
```



Checking Stationarity using ADF/KPSS

KPSS Test

- The null hypothesis is that the data are stationary, and we look for evidence that the null hypothesis is false.
- Null Hypothesis: The process is trend stationary.
- Alternate Hypothesis: The series has a unit root (series is not stationary)
- Small p-values (e.g., less than 0.05) suggest that differencing is required

```
def check_stationarity_KPSSTest(ts):
  kpsstest = kpss(ts)
  kpss_output = pd.Series(kpsstest[0:3], index=['Test Statistic','p-value','Lags Used'])
  for key,value in kpsstest[3].items():
    kpss_output['Critical Value (%s)'%key] = value
```

```
  print(kpss_output)
```

```
check_stationarity_KPSSTest(Energy_source_NaturalGas_New['Log'])
```

```
Test Statistic             1.914572
p-value                    0.010000
Lags Used                 18.000000
Critical Value (10%)       0.347000
Critical Value (5%)        0.463000
Critical Value (2.5%)      0.574000
Critical Value (1%)        0.739000
dtype: float64
```

ADF Test

- Null Hypothesis: The series has a unit root (value of a =1)
- Alternate Hypothesis: The series has no unit root.

```
def check_stationarity_ADFTest(ts):
  adftest = adfuller(ts)
  adfoutput = pd.Series(adftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number
  for key,value in adftest[4].items():
    adfoutput['Critical Value (%s)'%key] = value
  print (adfoutput)
```

```
check_stationarity_ADFTest(Energy_source_NaturalGas_New['Log'])
```

```
Test Statistic                  -0.653386
p-value                          0.858457
#Lags Used                      15.000000
Number of Observations Used    414.000000
Critical Value (1%)             -3.446244
Critical Value (5%)             -2.868547
Critical Value (10%)            -2.570502
dtype: float64
```

Result: Both test suggest that timeseries is not stationary

- This clears our understanding that seasonality of constant magnitude variation.
- We need to check for stationarity data using KPSS/ADF test and visual inspection.

Splitting data into train,test and forecast

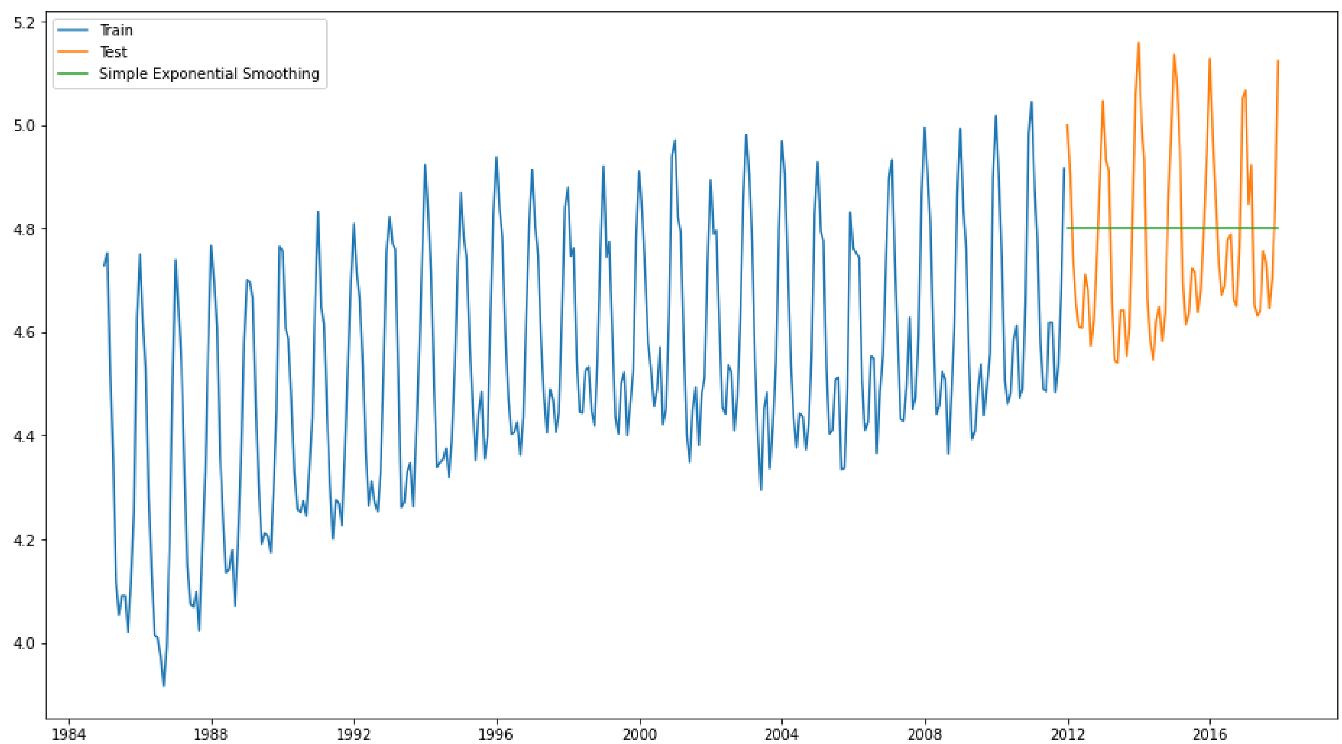
- Train-1985-2011
- Test- 2012-2017
- Forecast- 2018-2020

```
train= Energy_source_NaturalGas_New.iloc[:324]
test= Energy_source_NaturalGas_New.iloc[324:396]
forecast_validate= Energy_source_NaturalGas_New.iloc[396:430]
```

Exopential smoothing methods for forecasting

Simple Exponential Smoothing Method

```
sesModel = SimpleExpSmoothing(train['Log']).fit(smoothing_level=0.6,optimized=True)
ses_forecast_actual = sesModel.predict(start=test.index[0], end=test.index[-1])
fig = plt.figure(figsize = (16,9))
plt.plot(train.index, train['Log'], label='Train')
plt.plot(test.index, test['Log'], label='Test')
plt.plot(ses_forecast_actual.index, ses_forecast_actual, label='Simple Exponential Smoothing'
plt.legend(loc='best')
plt.show()
```




```
#RSME Value
rmse= sqrt(mean_squared_error(test['Log'],ses_forecast_actual))
```

```
print(rmse)
```

    0.17293012298277508

```
#Back transformed Forecast RMSE Check
btransformed= np.exp(ses_forecast_actual)
rmse= sqrt(mean_squared_error(test['Value'],btransformed))
print(rmse)
```

    21.778563803332002

Holt-Winter's Additive Model

```
holtModel = ExponentialSmoothing(train['Log'],trend='additive',seasonal='additive', seasonal_
holt_forecast_actual = holtModel.predict(start=test.index[0], end=test.index[-1])
btransformed_holt= np.exp(holt_forecast_actual)
fig = plt.figure(figsize = (16,9))
plt.plot(train.index, train['Value'], label='Train')
plt.plot(test.index, test['Value'], label='Test')
plt.plot(holt_forecast_actual.index, btransformed_holt, label='Holt-Winters')
plt.xlabel("Year")
plt.ylabel("Emission Quantity (in million metric tons)")
plt.legend(loc='best')
plt.show()
```

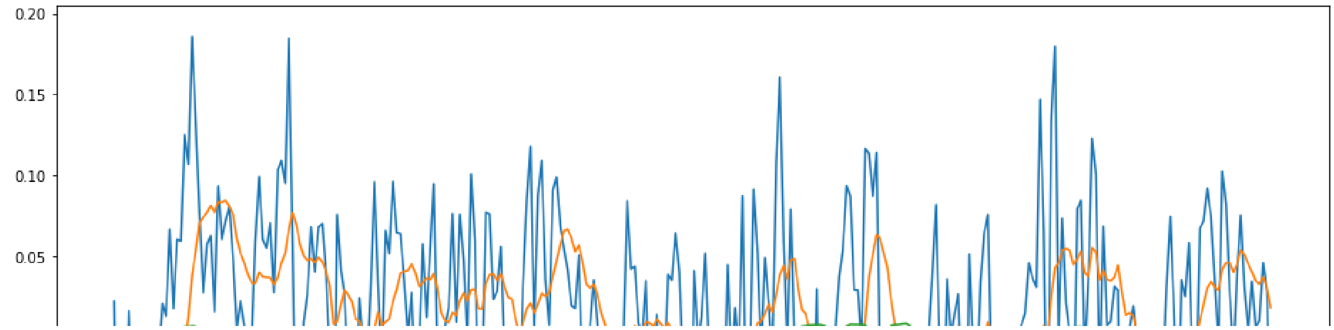

```
#RSME Value
rmse= sqrt(mean_squared_error(test['Log'],holt_forecast_actual))
print(rmse)
```

    0.07398972227047594

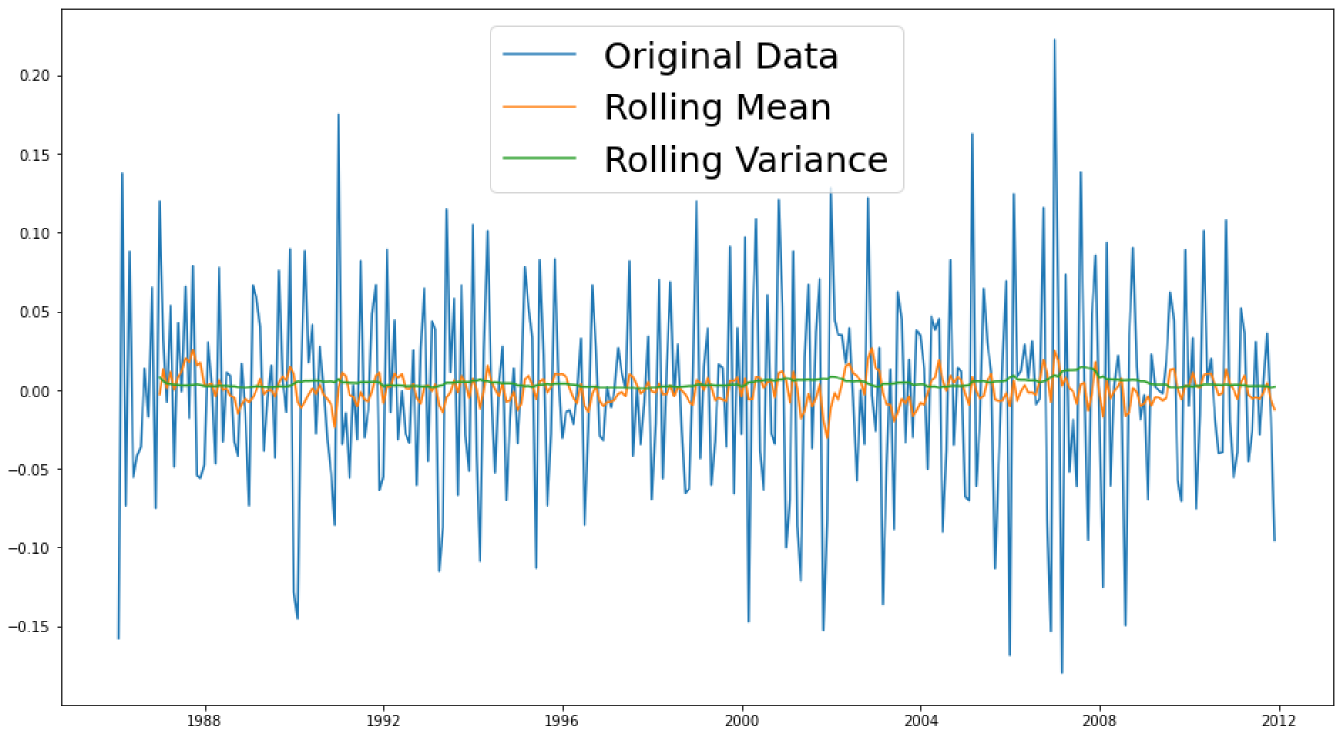
```
#Back transformin
rmse= sqrt(mean_squared_error(test['Value'],btransformed_holt))
print(rmse)
```

    8.897380882833925

```
train['seasonal_difference']=train['Log']-train['Log'].shift(12)
train['seasonal_difference'].dropna(inplace=True)
train['Rmean'] = train['seasonal_difference'].rolling(12).mean()
train['Rstdv'] = train['seasonal_difference'].rolling(12).var()
train['Rstdv'].replace(np.nan,0)
train['Rmean'].replace(np.nan,0)
fig = plt.figure(figsize = (16,9))
plt.plot(train.index,train['seasonal_difference'],label='Original Data')
plt.plot(train.index,train['Rmean'],label='Rolling Mean')
plt.plot(train.index,train['Rstdv'],label='Rolling Variance')
plt.legend(loc='best', fontsize = 25)
plt.show()
```




```
train['seasonal_first_difference']=train['seasonal_difference']-train['seasonal_difference'].
train["seasonal_first_difference"].dropna(inplace=True)
train['Rmean'] = train['seasonal_first_difference'].rolling(12).mean()
train['Rstdv'] = train['seasonal_first_difference'].rolling(12).var()
train['Rstdv'].replace(np.nan,0)
train['Rmean'].replace(np.nan,0)
fig = plt.figure(figsize = (16,9))
plt.plot(train.index,train['seasonal_first_difference'],label='Original Data')
plt.plot(train.index,train['Rmean'],label='Rolling Mean')
plt.plot(train.index,train['Rstdv'],label='Rolling Variance')
plt.legend(loc='best', fontsize = 25)
plt.show()
```



```
check_stationarity_KPSSTest(train['seasonal_first_difference'].dropna())
```

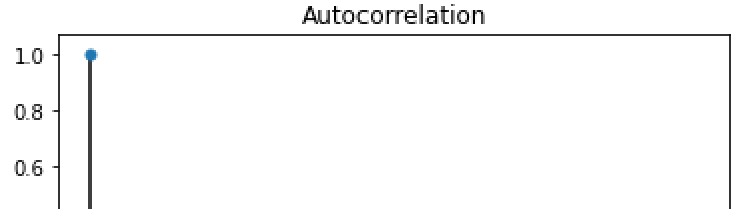
```
Test Statistic            0.035671
p-value                   0.100000
Lags Used                16.000000
Critical Value (10%)      0.347000
Critical Value (5%)       0.463000
Critical Value (2.5%)     0.574000
Critical Value (1%)       0.739000
dtype: float64
```

```
check_stationarity_ADFTest(train['seasonal_first_difference'].dropna())
```

```
Test Statistic                -7.187493e+00
p-value                        2.554227e-10
#Lags Used                     1.600000e+01
Number of Observations Used    2.940000e+02
Critical Value (1%)           -3.452790e+00
Critical Value (5%)           -2.871422e+00
Critical Value (10%)          -2.572035e+00
dtype: float64
```

Result: The results of ADF and KPSS test along with visual inspection using Graph suggest that after seasonal differencing followed by first order differencing we attain stationarity of times series.

```
plotACF(train['seasonal_first_difference'].dropna())
plotPACF(train['seasonal_first_difference'].dropna())
```



From these above ACF and PACF plot we can find the values of non-seasonal component and Seasonal Component of ARIMA. Here our findings suggest that p=2,d=1,q=0 and P=0,D=1,Q=1,m=12 would be appropriate to train the SARIMA Model.

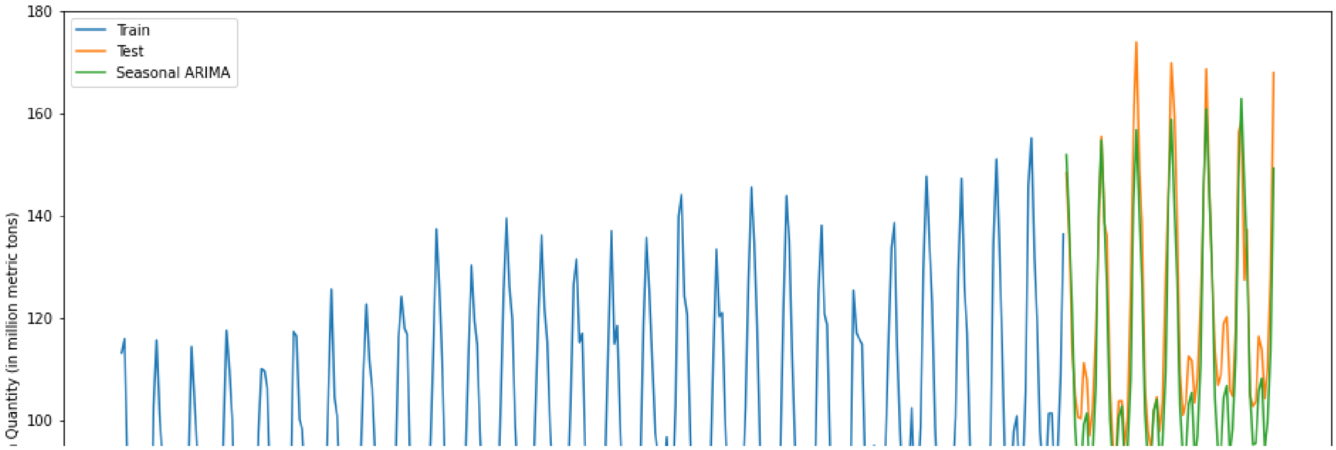
```
import statsmodels.api as sm
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

Partial Autocorrelation

```
SARIMA_Model=SARIMAX(train['Log'], order=(2, 1, 0), seasonal_order=(0, 1, 1, 12), enforce_inv
results = SARIMA_Model.fit()
forecast_sarima = results.predict(start=test.index[0], end=test.index[-1])
btransformed_arima= np.exp(forecast_sarima)
rmse= sqrt(mean_squared_error(test['Value'],btransformed_arima))
print(rmse)
```

```
9.668135516931324
```

```
fig = plt.figure(figsize = (16,9))
plt.plot(train.index, train['Value'], label='Train')
plt.plot(test.index, test['Value'], label='Test')
plt.plot(forecast_sarima.index, btransformed_arima, label='Seasonal ARIMA')
plt.xlabel("Year")
plt.ylabel("Emission Quantity (in million metric tons)")
plt.legend(loc='best')
plt.show()
```




```
rmse= sqrt(mean_squared_error(test['Log'],forecast_sarima))
print(rmse)
```

    0.08305653785823992

```
#Back transformation
rmse= sqrt(mean_squared_error(test['Value'],btransformed_arima))
print(rmse)
```

    9.668135516931324

On the comparing the RMSE values of Simple Exponential Smoothing Method, Holt-Winter Additive Method and Seasonal ARIMA Model(base model defined by our own understanding of p,d,q & P,D,Q Values) we can say the Holt Winter's Prediction on test data is more appropriate.

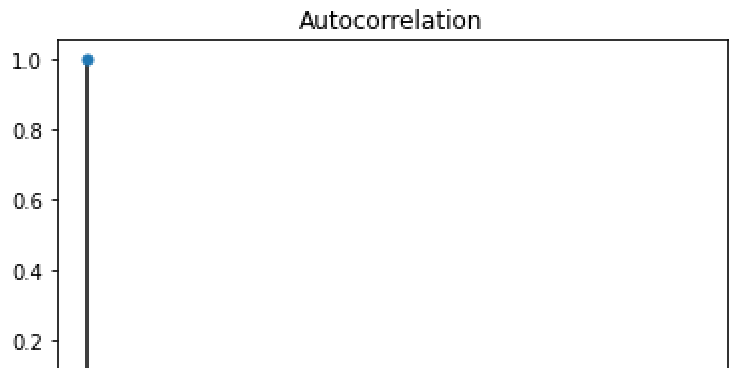
We can also check the residual values of the above trained model to confirm that there is some more information left in the residual.

```
#For SARIMA
fittedValues_sarima = results.predict(start=train.index[0], end=train.index[-1])
btransformed_fitted_sarima= np.exp(fittedValues_sarima)
residuals_sarima= train['Value']- btransformed_fitted_sarima
```

```
print(residuals_sarima.mean())
```

    -2.7922287462396547

```
plotACF(residuals_sarima)
```



Conclusions:

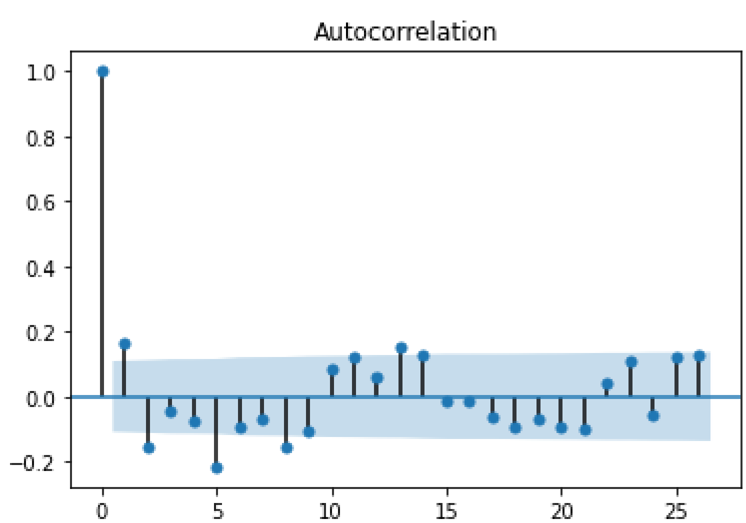
- From above 2 results we can see that mean of residuals is not close to zero, although residuals are not auto-correlated.
- Violation of this one property in residual diagnostics suggest that some more information is left in the timeseries.
- This can be the plausible reason that Holt Winter's Model is giving more accurate forecast on test data.
- We need to find more appropriate values of SARIMA parameters to attain more accurate forecast.

```
#For Holt-Winter Model
fittedValues_holt = holtModel.predict(start=train.index[0], end=train.index[-1])
btransformed_fitted_holt= np.exp(fittedValues_holt)
residuals_holt= train['Value']- btransformed_fitted_holt
```

```
print(residuals_holt.mean())
```

```
-0.12371492009137486
```

```
plotACF(residuals_holt)
```



Using GridSearch Method to find the best fit of SARIMA parameters

```
import itertools

p=q=range(0,3)
d=range(1,2)

pdq=list(itertools.product(p,d,q))
seasonal_pdq=[(x[0],x[1],x[2],12) for x in list(itertools.product(p,d,q))]
print(seasonal_pdq)
print(pdq)
for i in range(0,9):
  for j in range(0,9):
    print('SARIMAX: {} X {}'.format(pdq[j],seasonal_pdq[j]))
```

```
SARIMAX: (1, 1, 1) X (1, 1, 1, 12)
SARIMAX: (1, 1, 2) X (1, 1, 2, 12)
SARIMAX: (2, 1, 0) X (2, 1, 0, 12)
SARIMAX: (2, 1, 1) X (2, 1, 1, 12)
SARIMAX: (2, 1, 2) X (2, 1, 2, 12)
SARIMAX: (0, 1, 0) X (0, 1, 0, 12)
SARIMAX: (0, 1, 1) X (0, 1, 1, 12)
SARIMAX: (0, 1, 2) X (0, 1, 2, 12)
SARIMAX: (1, 1, 0) X (1, 1, 0, 12)
SARIMAX: (1, 1, 1) X (1, 1, 1, 12)
SARIMAX: (1, 1, 2) X (1, 1, 2, 12)
SARIMAX: (2, 1, 0) X (2, 1, 0, 12)
SARIMAX: (2, 1, 1) X (2, 1, 1, 12)
SARIMAX: (2, 1, 2) X (2, 1, 2, 12)

SARIMAX: (0, 1, 0) X (0, 1, 0, 12)
SARIMAX: (0, 1, 1) X (0, 1, 1, 12)
SARIMAX: (0, 1, 2) X (0, 1, 2, 12)
SARIMAX: (1, 1, 0) X (1, 1, 0, 12)
SARIMAX: (1, 1, 1) X (1, 1, 1, 12)
SARIMAX: (1, 1, 2) X (1, 1, 2, 12)
SARIMAX: (2, 1, 0) X (2, 1, 0, 12)
SARIMAX: (2, 1, 1) X (2, 1, 1, 12)
SARIMAX: (2, 1, 2) X (2, 1, 2, 12)
SARIMAX: (0, 1, 0) X (0, 1, 0, 12)
SARIMAX: (0, 1, 1) X (0, 1, 1, 12)
SARIMAX: (0, 1, 2) X (0, 1, 2, 12)
SARIMAX: (1, 1, 0) X (1, 1, 0, 12)
SARIMAX: (1, 1, 1) X (1, 1, 1, 12)
SARIMAX: (1, 1, 2) X (1, 1, 2, 12)
SARIMAX: (2, 1, 0) X (2, 1, 0, 12)
SARIMAX: (2, 1, 1) X (2, 1, 1, 12)
SARIMAX: (2, 1, 2) X (2, 1, 2, 12)
SARIMAX: (0, 1, 0) X (0, 1, 0, 12)
SARIMAX: (0, 1, 1) X (0, 1, 1, 12)
SARIMAX: (0, 1, 2) X (0, 1, 2, 12)
SARIMAX: (1, 1, 0) X (1, 1, 0, 12)
SARIMAX: (1, 1, 1) X (1, 1, 1, 12)
SARIMAX: (1, 1, 2) X (1, 1, 2, 12)
```

```
SARIMAX: (2, 1, 0) X (2, 1, 0, 12)
SARIMAX: (2, 1, 1) X (2, 1, 1, 12)
SARIMAX: (2, 1, 2) X (2, 1, 2, 12)
SARIMAX: (0, 1, 0) X (0, 1, 0, 12)
SARIMAX: (0, 1, 1) X (0, 1, 1, 12)
SARIMAX: (0, 1, 2) X (0, 1, 2, 12)
SARIMAX: (1, 1, 0) X (1, 1, 0, 12)
SARIMAX: (1, 1, 1) X (1, 1, 1, 12)
SARIMAX: (1, 1, 2) X (1, 1, 2, 12)
SARIMAX: (2, 1, 0) X (2, 1, 0, 12)
SARIMAX: (2, 1, 1) X (2, 1, 1, 12)
SARIMAX: (2, 1, 2) X (2, 1, 2, 12)
SARIMAX: (0, 1, 0) X (0, 1, 0, 12)
SARIMAX: (0, 1, 1) X (0, 1, 1, 12)
SARIMAX: (0, 1, 2) X (0, 1, 2, 12)
SARIMAX: (1, 1, 0) X (1, 1, 0, 12)
SARIMAX: (1, 1, 1) X (1, 1, 1, 12)
SARIMAX: (1, 1, 2) X (1, 1, 2, 12)
SARIMAX: (2, 1, 0) X (2, 1, 0, 12)
SARIMAX: (2, 1, 1) X (2, 1, 1, 12)
```

```
metric_aic_dict=dict()

for pm in pdq:
  for pm_seasonal in seasonal_pdq:
    model=sm.tsa.statespace.SARIMAX(train['Log'],order=pm, seasonal_order=pm_seasonal, enforc
    model_aic=model.fit()
    print('ARIMA{} X{}12-AIC:{}'.format(pm,pm_seasonal,model_aic.aicc))
    metric_aic_dict.update({(pm,pm_seasonal):model_aic.aicc})
```

```
ARIMA(0, 1, 2) X(1, 1, 1, 12)12-AIC:-1015.9876178774625
ARIMA(0, 1, 2) X(1, 1, 2, 12)12-AIC:-1013.9830228816182
ARIMA(0, 1, 2) X(2, 1, 0, 12)12-AIC:-996.2211697157935

ARIMA(0, 1, 2) X(2, 1, 1, 12)12-AIC:-1014.2431021812382
ARIMA(0, 1, 2) X(2, 1, 2, 12)12-AIC:-1011.7356404634627
ARIMA(1, 1, 0) X(0, 1, 0, 12)12-AIC:-848.4993437674641
ARIMA(1, 1, 0) X(0, 1, 1, 12)12-AIC:-972.5765479703389
ARIMA(1, 1, 0) X(0, 1, 2, 12)12-AIC:-970.5259804062385
ARIMA(1, 1, 0) X(1, 1, 0, 12)12-AIC:-913.9329452251354
ARIMA(1, 1, 0) X(1, 1, 1, 12)12-AIC:-970.5190674459894
ARIMA(1, 1, 0) X(1, 1, 2, 12)12-AIC:-968.8164954027027
ARIMA(1, 1, 0) X(2, 1, 0, 12)12-AIC:-948.1771102933894
ARIMA(1, 1, 0) X(2, 1, 1, 12)12-AIC:-970.5787811605505
ARIMA(1, 1, 0) X(2, 1, 2, 12)12-AIC:-968.3382521372242
ARIMA(1, 1, 1) X(0, 1, 0, 12)12-AIC:-913.6242970391042
ARIMA(1, 1, 1) X(0, 1, 1, 12)12-AIC:-1022.8218120936585
ARIMA(1, 1, 1) X(0, 1, 2, 12)12-AIC:-1020.7733067224342
ARIMA(1, 1, 1) X(1, 1, 0, 12)12-AIC:-974.2932502186263
ARIMA(1, 1, 1) X(1, 1, 1, 12)12-AIC:-1020.7702623322714
ARIMA(1, 1, 1) X(1, 1, 2, 12)12-AIC:-1018.9254854011411
ARIMA(1, 1, 1) X(2, 1, 0, 12)12-AIC:-1001.6954887873329
ARIMA(1, 1, 1) X(2, 1, 1, 12)12-AIC:-1019.9529736482644
ARIMA(1, 1, 1) X(2, 1, 2, 12)12-AIC:-1017.3384428492866
ARIMA(1, 1, 2) X(0, 1, 0, 12)12-AIC:-911.9273303806949
ARIMA(1, 1, 2) X(0, 1, 1, 12)12-AIC:-1021.9923367733027
```

```
ARIMA(1, 1, 2) X(0, 1, 2, 12)12-AIC:-1015.1832741827278
ARIMA(1, 1, 2) X(1, 1, 0, 12)12-AIC:-972.5035624476868
ARIMA(1, 1, 2) X(1, 1, 1, 12)12-AIC:-1019.9085025716196
ARIMA(1, 1, 2) X(1, 1, 2, 12)12-AIC:-1018.0664358475071
ARIMA(1, 1, 2) X(2, 1, 0, 12)12-AIC:-999.8690358822765
ARIMA(1, 1, 2) X(2, 1, 1, 12)12-AIC:-1019.0076390477363
ARIMA(1, 1, 2) X(2, 1, 2, 12)12-AIC:-1015.1100962124694
ARIMA(2, 1, 0) X(0, 1, 0, 12)12-AIC:-891.358769162397
ARIMA(2, 1, 0) X(0, 1, 1, 12)12-AIC:-994.1271600886123
ARIMA(2, 1, 0) X(0, 1, 2, 12)12-AIC:-992.2400204170685
ARIMA(2, 1, 0) X(1, 1, 0, 12)12-AIC:-951.5251525300581
ARIMA(2, 1, 0) X(1, 1, 1, 12)12-AIC:-992.2155868644793
ARIMA(2, 1, 0) X(1, 1, 2, 12)12-AIC:-990.3458576445252
ARIMA(2, 1, 0) X(2, 1, 0, 12)12-AIC:-973.9856064236155
ARIMA(2, 1, 0) X(2, 1, 1, 12)12-AIC:-990.8864819532531
ARIMA(2, 1, 0) X(2, 1, 2, 12)12-AIC:-988.3226978785455
ARIMA(2, 1, 1) X(0, 1, 0, 12)12-AIC:-911.6248288382587
ARIMA(2, 1, 1) X(0, 1, 1, 12)12-AIC:-1021.5711409237442
ARIMA(2, 1, 1) X(0, 1, 2, 12)12-AIC:-1019.5058998632925
ARIMA(2, 1, 1) X(1, 1, 0, 12)12-AIC:-972.2808728981795
ARIMA(2, 1, 1) X(1, 1, 1, 12)12-AIC:-1019.5059512565675
ARIMA(2, 1, 1) X(1, 1, 2, 12)12-AIC:-1017.6706338473398
ARIMA(2, 1, 1) X(2, 1, 0, 12)12-AIC:-999.7624283224392
ARIMA(2, 1, 1) X(2, 1, 1, 12)12-AIC:-1007.6595176156882
ARIMA(2, 1, 1) X(2, 1, 2, 12)12-AIC:-1016.0716953225149
ARIMA(2, 1, 2) X(0, 1, 0, 12)12-AIC:-916.4462109129454
ARIMA(2, 1, 2) X(0, 1, 1, 12)12-AIC:-1019.7835763780735
ARIMA(2, 1, 2) X(0, 1, 2, 12)12-AIC:-1017.6734308671591
ARIMA(2, 1, 2) X(1, 1, 0, 12)12-AIC:-971.6774266058261
ARIMA(2, 1, 2) X(1, 1, 1, 12)12-AIC:-1017.6624939239327
ARIMA(2, 1, 2) X(1, 1, 2, 12)12-AIC:-1015.7121797015795
ARIMA(2, 1, 2) X(2, 1, 0, 12)12-AIC:-998.068757429674
ARIMA(2, 1, 2) X(2, 1, 1, 12)12-AIC:-1016.6617115565289
ARIMA(2, 1, 2) X(2, 1, 2, 12)12-AIC:-1013.9182057582174
```

```
{k: v for k,v in sorted(metric_aic_dict.items(),key=lambda x:x[1])}
```

```
((0, 1, 2), (1, 1, 0, 12)): -973.1669823066853,
((0, 1, 2), (1, 1, 1, 12)): -1015.9876178774625,
((0, 1, 2), (1, 1, 2, 12)): -1013.9830228816182,
((0, 1, 2), (2, 1, 0, 12)): -996.2211697157935,
((0, 1, 2), (2, 1, 1, 12)): -1014.2431021812382,
((0, 1, 2), (2, 1, 2, 12)): -1011.7356404634627,
((1, 1, 0), (0, 1, 0, 12)): -848.4993437674641,
((1, 1, 0), (0, 1, 1, 12)): -972.5765479703389,
((1, 1, 0), (0, 1, 2, 12)): -970.5259804062385,
((1, 1, 0), (1, 1, 0, 12)): -913.9329452251354,
((1, 1, 0), (1, 1, 1, 12)): -970.5190674459894,
((1, 1, 0), (1, 1, 2, 12)): -968.8164954027027,
((1, 1, 0), (2, 1, 0, 12)): -948.1771102933894,
((1, 1, 0), (2, 1, 1, 12)): -970.5787811605505,
((1, 1, 0), (2, 1, 2, 12)): -968.3382521372242,
((1, 1, 1), (0, 1, 0, 12)): -913.6242970391042,
((1, 1, 1), (0, 1, 1, 12)): -1022.8218120936585,
((1, 1, 1), (0, 1, 2, 12)): -1020.7733067224342,
((1, 1, 1), (1, 1, 0, 12)): -974.2932502186263,
((1, 1, 1), (1, 1, 1, 12)): -1020.7702623322714,
((1, 1, 1), (1, 1, 2, 12)): -1018.9254854011411,
```

```
((1, 1, 1), (2, 1, 0, 12)): -1001.6954887873329,
((1, 1, 1), (2, 1, 1, 12)): -1019.9529736482644,
((1, 1, 1), (2, 1, 2, 12)): -1017.3384428492866,
((1, 1, 2), (0, 1, 0, 12)): -911.9273303806949,
((1, 1, 2), (0, 1, 1, 12)): -1021.9923367733027,
((1, 1, 2), (0, 1, 2, 12)): -1015.1832741827278,
((1, 1, 2), (1, 1, 0, 12)): -972.5035624476868,
((1, 1, 2), (1, 1, 1, 12)): -1019.9085025716196,
((1, 1, 2), (1, 1, 2, 12)): -1018.0664358475071,
((1, 1, 2), (2, 1, 0, 12)): -999.8690358822765,
((1, 1, 2), (2, 1, 1, 12)): -1019.0076390477363,
((1, 1, 2), (2, 1, 2, 12)): -1015.1100962124694,
((2, 1, 0), (0, 1, 0, 12)): -891.358769162397,
((2, 1, 0), (0, 1, 1, 12)): -994.1271600886123,
((2, 1, 0), (0, 1, 2, 12)): -992.2400204170685,
((2, 1, 0), (1, 1, 0, 12)): -951.5251525300581,
((2, 1, 0), (1, 1, 1, 12)): -992.2155868644793,
((2, 1, 0), (1, 1, 2, 12)): -990.3458576445252,
((2, 1, 0), (2, 1, 0, 12)): -973.9856064236155,
((2, 1, 0), (2, 1, 1, 12)): -990.8864819532531,
((2, 1, 0), (2, 1, 2, 12)): -988.3226978785455,
((2, 1, 1), (0, 1, 0, 12)): -911.6248288382587,
((2, 1, 1), (0, 1, 1, 12)): -1021.5711409237442,
((2, 1, 1), (0, 1, 2, 12)): -1019.5058998632925,
((2, 1, 1), (1, 1, 0, 12)): -972.2808728981795,
((2, 1, 1), (1, 1, 1, 12)): -1019.5059512565675,
((2, 1, 1), (1, 1, 2, 12)): -1017.6706338473398,
((2, 1, 1), (2, 1, 0, 12)): -999.7624283224392,
((2, 1, 1), (2, 1, 1, 12)): -1007.6595176156882,
((2, 1, 1), (2, 1, 2, 12)): -1016.0716953225149,
((2, 1, 2), (0, 1, 0, 12)): -916.4462109129454,
((2, 1, 2), (0, 1, 1, 12)): -1019.7835763780735,
((2, 1, 2), (0, 1, 2, 12)): -1017.6734308671591,
((2, 1, 2), (1, 1, 0, 12)): -971.6774266058261,
((2, 1, 2), (1, 1, 1, 12)): -1017.6624939239327,

((2, 1, 2), (1, 1, 2, 12)): -1015.7121797015795,
((2, 1, 2), (2, 1, 0, 12)): -998.068757429674,
((2, 1, 2), (2, 1, 1, 12)): -1016.6617115565289,
((2, 1, 2), (2, 1, 2, 12)): -1013.0182057582174}
```

```
SARIMA_Model2=SARIMAX(train['Log'], order=(1, 1, 1), seasonal_order=(0, 1, 1, 12), enforce_in
results2 = SARIMA_Model2.fit()
forecast_sarima2 = results2.predict(start=test.index[0], end=test.index[-1])
btransformed_arima2= np.exp(forecast_sarima2)
rmse2= sqrt(mean_squared_error(test['Value'],btransformed_arima2))
print(rmse2)
```

```
9.314761072642844
```

```
SARIMA_Model1=SARIMAX(train['Log'], order=(1, 1, 2), seasonal_order=(0, 1, 1, 12), enforce_in
results1 = SARIMA_Model1.fit()
forecast_sarima1 = results1.predict(start=test.index[0], end=test.index[-1])
btransformed_arima1= np.exp(forecast_sarima1)
rmse1= sqrt(mean_squared_error(test['Value'],btransformed_arima1))
print(rmse1)
```

```
9.116830626422855
```

```
fig = plt.figure(figsize = (16,9))
plt.plot(train.index, train['Value'], label='Train')
plt.plot(test.index, test['Value'], label='Test')
plt.plot(forecast_sarima.index, btransformed_arima, label='Seasonal ARIMA (2,1,0)(0,1,1)')
plt.plot(forecast_sarima.index, btransformed_arima1, label='Seasonal ARIMA (1,1,2)(0,1,1)')
plt.plot(holt_forecast_actual.index, btransformed_holt, label='Holt-Winters')
plt.xlabel("Year")
plt.ylabel("Emission Quantity (in million metric tons)")
plt.legend(loc='best')
plt.show()
```